AD-A197 217

1

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AFIT/CI/NR 88-168	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER DTIC FILE COPY	
4. TITLE (and Subtitle) CREATING STAND-ALONE SMALLTALK APPLICATIONS	5. TYPE OF REPORT & PERIOD COVERED MS THESIS
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) PAUL D. GILBERT	8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS AFIT STUDENT AT: UNIVERSITY OF ILLINOIS	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS	12. REPORT DATE 1988
	13. NUMBER OF PAGES 25
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) AFIT/NR Wright-Patterson AFB OH 45433-6583	15. SECURITY CLASS. (of this report) UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

DISTRIBUTED UNLIMITED: APPROVED FOR PUBLIC RELEASE

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

SAME AS REPORT

18. SUPPLEMENTARY NOTES

Approved for Public Release: IAW AFR 190-1
LYNN E. WOLAVER Lynn Wolaver 8 Aug 88
Dean for Research and Professional Development
Air Force Institute of Technology
Wright-Patterson AFB OH 45433-6583

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

ATTACHED

DTIC
S ELECTE D
AUG 1 9 1988
H

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

88 8 16 028

Creating Stand-Alone Smalltalk Applications

Paul D. Gilbert

June 15, 1988

Abstract

Smalltalk-80 is a large programming environment. Developing an application program in Smalltalk involves re-using or modifying parts of the environment and adding more Smalltalk code to the environment. An application is thus dependent on the environment; the entire environment is present when executing an application program even though the application might rely on only a small portion of the environment. This paper describes the process of separating an *application from the* Smalltalk environment so that the application can be executed as a stand-alone program unit. It also describes the Smalltalk system tracer, an essential tool for application tracing, and image inspector, a tool for examining a Smalltalk image that is saved on disk.

1 Introduction

The Smalltalk-80 programming environment offers many advantages. The environment includes a sophisticated debugger, a text editor, a compiler to translate Smalltalk code into interpretable code, and a number of other utilities. Access to these programs is provided through a graphical user-interface based on over-lapping windows. Each window represents a task, only one of which is active at any time. The user selects the active window using a mouse. This user interface provides a highly integrated environment which allows a programmer to quickly switch between programming, compiling, testing and debugging activities.

All of the code to implement the Smalltalk environment is accessible to the user. Smalltalk code is organized into classes. Classes include methods, the Smalltalk equivalent of functions. A program called the browser presents the classes and their methods grouped by category for easy access. Developing a Smalltalk application involves re-using or modifying existing classes as well as adding new classes. An application is therefore dependent on the environment and difficult to distinguish from the environment.

This integrated environment is an advantage when developing an application but becomes a disadvantage when delivering a finished software product. One problem is size. The entire Smalltalk environment (which is over 1 MByte) and the Smalltalk interpreter must be present to execute an application. This restricts the number of machines on which a Smalltalk application can run. Another problem is that users of a Smalltalk application have too much access to the underlying system. This presents a reliability and security problem.

To create smaller applications, we need some way to identify the parts of the environment that are actually needed for an application and strip the rest of the environment away. I'll refer to this process as application tracing, named after the system tracing capability provided in most Smalltalk implementations. The next section of this paper presents an overview of application tracing. Section 3 presents a more detailed discussion of the Tektronix system tracer. Section 4 discusses the program I wrote to do application tracing. Section 5 discusses image inspector, a tool I developed for debugging the application tracer.

2 Overview of Application Tracing

A Smalltalk environment, also called an image, is made up entirely of objects. For example, there are objects to represent classes, methods, and variables. Saving a Smalltalk environment requires that all of the accessible objects be written to secondary storage. The format of the image is implementation dependent.

For
✓
A-1

One way to copy objects of an environment to secondary storage is by executing the system tracer (sometimes called the system cloner). The system tracer creates a new image from the image that is currently executing by writing a copy of every accessible object to an image file. The new image can be the same as the image in memory or it can be a transformation of the image in memory.

The system tracer thus provides an obvious vehicle for implementing application tracing. The idea is to transform the image in memory by eliminating objects that are not needed for some application. The system tracer can then be used to copy the image to secondary storage.

The question then is, how do we perform such a transformation? The Tektronix implementation of system tracer provides two capabilities for transforming an image. One capability is to create an image that excludes some user-specified set of classes. Excluding a class also eliminates the methods implemented in the class and all instances of the class. This capability might be useful when re-implementing a class. After testing a new implementation, a user could create a new image that excludes the old implementation of the class. The capability to exclude classes, however, was not intended and is not effective for application tracing. An application developer doesn't know what classes will be needed. He may know, for example, that the application uses class C but not know which classes are referenced directly or indirectly by class C. In addition, a finer granularity is needed; we need the capability to delete methods and other individual objects.

Another option for creating a transformed image is a process called winnowing. The purpose of winnowing is to remove unreferenced methods (i.e. those for which there is no sender). During the winnowing process, the system tracer makes a pass through each class in the Smalltalk environment, eliminating methods whose selectors are not sent by any other method. Then it makes another pass, picking up those methods that are now unreferenced due to the preceding deletions. The process continues for the number of passes specified by the user. Winnowing is essentially an implementation of the graph algorithm known as a topological sort.

At first glance, winnowing might seem a good way to eliminate methods that are not needed for some application. Winnowing, however, does not eliminate methods that are invoked in cycles. For example, if method m1 references method m2 and m2 references m1, neither method will be deleted even though they may not be needed for some application.

A better way to implement application tracing is to identify methods that are *referenced* rather than searching for methods that are *unreferenced*. Conceptually this is a graph problem. Imagine a graph in which each node represents a method in the Smalltalk environment. Suppose we connect the nodes by adding a directed arc from a node m1 to a node m2 if and only if method m1 invokes method m2. If we constructed such a graph, identifying the methods needed by an application would be simple. We could traverse the graph beginning with the "initial method" (i.e. the first method to be invoked when the image is executed). The initial method is analogous to the main program in a block-structured language like C or Pascal; it specifies the complete application at a high level. The only methods needed for our application would be those reached by a traversal from the initial method.

Unfortunately, we cannot construct such a graph. Smalltalk methods do not directly invoke other methods; instead they send messages. The actual method that will respond to a message is not determined until run time. There may be many methods that respond to a given message. The graph concept, however, can still be applied. Each node still represents a method. We just need more arcs. If a method m1 sends message m, we need an arc from node m1 to every node that implements message m. After constructing such a graph, we can still identify all of the methods needed for an application by performing a graph traversal that begins with the initial method.

In practice, however, saving every implementation of every reachable message doesn't work much better than winnowing; many unnecessary messages are preserved. For example, I traced an application that did nothing but write some text on the screen and then quit. The resulting image was about 900K bytes and included about 25K objects. Only about 10% of the Smalltalk environment was eliminated in generating this application.

One reason this happens is that Smalltalk uses "standard protocols" i.e. the same message names are used to implement similar functions in many classes. For example, most classes implement the printOn: message, which prints a textual representation of an object. Any reasonable application makes use of standard protocols and hence could potentially invoke most of the methods in the Smalltalk environment.

Limiting the size of an image requires that we limit the number of connections in our graph. Rather than including every implementation of some message, we need a way to identify the implementations that are really needed. The type system, developed by Professor Johnson and Justin Graver, provides this capability [JG87]. For each message send, we can examine the type of the receiver, identify the classes included in this type, and then include only the message implementations in those classes. Using this approach, I generated an application that does simple arithmetic. The resulting image was about 48K bytes and included 1,640 objects.

OFFSET	SIZE	DESCRIPTION
0	4	size of image in bytes (excluding image header)
4	4	number of objects (according to program comments, but seems to be unused)
8	2	a code to identify the machine and processor for which the image was written
10	2	number of known objects
12	8	offset and size of gradefile # 1
20	8	offset and size of gradefile # 2
28	8	offset and size of gradefile # 3
36	8	offset and size of gradefile # 4
44	8	offset and size of gradefile # 5
52	8	offset and size of gradefile # 6
60	8	offset and size of gradefile # 7
68	8	offset and size of gradefile # 8
76	8	offset and size of large objects file
84	4	number of large objects in file
88	88	table of known objects
176	336	seems to be unused

Table 1: Format of a Tektronix Image Header

3 System Tracer

The system tracer writes an image to secondary storage by copying objects from the image in memory. The Tektronix version is implemented by a class called SystemTracer. An image on disk begins with a 512-byte image header formatted as shown in table 1. The offsets and field sizes are a decimal number of bytes.

The table starting at file position 88 consists of an object-oriented pointer (OOP) to each "known object". An OOP is a 32-bit value that specifies the position of an object within the image file. "Known objects" (also referred to as "special objects") are those which the Smalltalk interpreter must be able to access in order to execute Smalltalk code. For example, one special object is the class SmallInteger. Given an object, the interpreter must be able to access the object's class. Objects other than small integers contain a class OOP. Since instances of SmallInteger do not contain a class OOP, the Smalltalk interpreter uses the known object table to access the class.

Following the 512-byte header are the objects. Each object begins on a

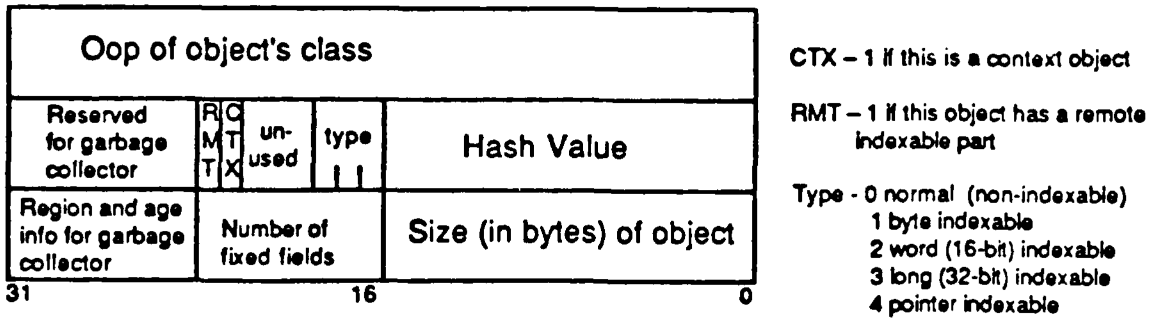
Oop of object's class
Reserved for garbage collector
RMT
CTX
un-used
type
Hash Value
Region and age info for garbage collector
Number of fixed fields
Size (in bytes) of object
31
16
0
CTX – 1 if this is a context object
RMT – 1 if this object has a remote indexable part
Type - 0 normal (non-indexable)
1 byte indexable
2 word (16-bit) indexable
3 long (32-bit) indexable
4 pointer indexable

Figure 1: Format of a Tektronix Object Header

(4-byte) word boundary and contains a 12-byte header organized as shown in figure 1[CW86]. This diagram reads from bottom to top. The first byte is data used by the generation scavenging scheme for garbage collection. See [Ung84] for a presentation of this algorithm. The second byte specifies the number of named instance variables. The next 16-bit field gives the size of the object in bytes. Then, following another byte that is used for garbage collection, is a flags byte. Next comes a 16-bit hash word and then the OOP of the object's class.

The object's header is followed by other fields of the object. Interpretation of these fields depends on the flags field. For example, an object with five instance variables and no indexable part would contain five 32-bit fields, one for each instance variable. Each 32-bit field would be either a SmallInteger value or an OOP. A SmallInteger is distinguished from an OOP by the high-order bit, which is 0 for a SmallInteger and 1 for an OOP.

As mentioned in the overview, the system tracer can be used to create a new image equivalent to the image in memory or it can create a transformation of the image in memory.

To create an image without transformation, evaluate the expression

```
SystemTracer writeClone.
```

This method will prompt and read a file name and then write the image to a file with the name input.

To create an image that excludes some set of classes, evaluate an expression of the form

```
SystemTracer writeCloneWithout: aSetOfClasses.
```

This method "clamps" out each class in **aSetOfClasses**. Clamping a class involves the following:

1. The class reference is removed from the system dictionary, **Smalltalk**.

2. The class is removed from its parent's list of subclasses.

3. The class is removed from the **SystemOrganization** global variable which identifies the classes in each class category. If a class category becomes empty, the category is removed from **SystemOrganization**.

Instances of a clamped class are not written to the new image. If an object that contains references to a clamped class is written to the new image, these references are changed to the OOP of the nil object.

To create an image using the winnowing process, send a message of the form

winnow: numberOfPasses

to an instance of **SystemTracer**. This message will make the specified number of passes through the system clamping out unreferenced methods. When winnowing completes, you must send the **writeClone** message to copy the image to disk.

3.1 Implementation of SystemTracer

As stated earlier, the Smalltalk environment consists entirely of objects. Every object references at least one other object. We can conceptualize an image as a directed graph where each node represents an object and each edge represents an object reference (i.e. an OOP). **SystemTracer** uses a graph traversal algorithm to identify the accessible objects. The traversal is implemented as a sequence of depth-first searches beginning with certain root objects. The root objects consist of the special objects, the symbols **#Smalltalk** and **#Processor**, and the global variable **Smalltalk**.

To create a new image, **SystemTracer** writes an image header and writes the objects visited during a depth-first traversal from the root objects. Any object which is not reachable from one of the root objects is not written to the image file.

A depth-first search of the object space is initiated by sending the message

trace: anObject

to an instance of **SystemTracer**. To understand what happens when **trace**: is invoked, let's consider an example. Suppose we are tracing a non-indexable object called A which has one instance variable called B. Object B is a non-indexable object with instance variable C. Object C is a string. The following graph represents this exampie.

$$\begin{tikzcd}
\text{A class} & \text{B class} & \text{C class} \\
\text{A} \arrow[u] \arrow[r] & \text{B} \arrow[u] \arrow[r] & \text{C} \arrow[u]
\end{tikzcd}$$

Objects are written to the new image file in the order that they are traced. When object A is traced, the first 8 bytes of the object header are added to the end of the image file being created. Next, the OOP that references A's class and the OOP that references object B must be written. OOPs in memory, however, do not correspond to OOPs in the new image. An OOP is calculated from its object's position in the file, but this position is not known until the object is traced. Therefore, after space is reserved for the remainder of object A, a trace is invoked on A's class and then on object B. Once these traces are complete, the OOPs for A's class and for object B are known. The image file is then re-positioned to 8 bytes after the start of object A. Then A's class OOP and the OOP for B are written.

Of course the trace of B will cause a trace on B's class and on object C. The trace of C will cause a trace on C's class. Therefore, the trace of object A cannot be completed until objects A class, B class, C class, B, and C have been traced. This example illustrates the necessity of a depth-first traversal to trace an image.

Different classes of objects have different formats for non-header data. We saw that for A, which was a non-indexable object, the non-header data was an oop for its instance variable. In contrast, C (of class **String**) is a byte-indexable object. C's non-header data therefore contains the bytes of its string value.

SystemTracer uses different methods to write the different object formats. To determine which method to invoke, the **trace**: method uses an instance variable called **writeDict**. **writeDict** is a dictionary that associates each class with the appropriate message selector for writing objects of that class. The **trace**: method sends the **perform:with**: message to invoke the correct writing method.

The writing method selectors are #**writeContext:**, #**writePointers:**, #**writeIndexablePointers:**, #**writeBytes:**, #**writeChars:**, #**writeWords:**, #**writeSet:**, #**writeIdentityDictionary:**, #**writeMethodDictionary:**, and #**writeProcess:**. Each method selected by these symbols sends a message of the form

new: obj
class: class
length: length
flags: flags
grade: grade
trace: traceBlock

write: writeBlock

to an instance of **SystemTracer**. This method, abbreviated as **new...write** in this paper, is executed for every object written to the image file. The **new...write** method writes the header data in the same format for all objects, but the non-header data varies. This variation is accounted for by the parameters **traceBlock** and **writeBlock**. The **traceBlock** contains code to trace objects referenced in the non-header data. The **writeBlock** contains code to write the non-header data.

The **new...write** method first writes the object header. Next, if the **traceBlock** is non-nil, the object's class is traced. Next the **traceBlock** is evaluated to trace other referenced objects. Once the object's class OOP and other referenced objects have been traced, their OOPs are known. The object's class OOP can then be written. Finally, the **writeBlock** is evaluated to write the non-header data.

A depth-first traversal algorithm generally requires some way of determining whether or not a node has already been visited. Otherwise, nodes would be visited multiple times. When tracing an image, multiple visits would mean multiple copies of the same object on an image file, which clearly would be a problem. Even worse, the traversal of a cyclic graph (like the Smalltalk object space) would not terminate.

To avoid these problems, **SystemTracer** uses an instance variable called **hashTable** to mark objects as "visited". The hash *value* associated with objects in the hash table is a "partial OOP", which can be easily transformed into an OOP. (The method **SystemTracer writeOopOf:on:** performs this transformation.) When **trace**: **anObject** is invoked, it checks the hash table. If **anObject** is already in the hash table, the trace terminates. If **anObject** is not in the hash table, the object is traced (i.e. the appropriate write message from **writeDict** is sent) and the object is placed in the hash table.

When **trace**: is called on an object of class **SmallInteger**, the **trace**: method simply returns. **SmallInteger** instances are not traced because they are not stored as separate objects. Instead, small integer values are simply written into an object in place of an OOP.

The hash table also plays a role in the clamping process. When an object is clamped, the object is placed into the hash table with a hash value equal to Clamped (a class variable with value -2). This serves two purposes. First, since a clamped object is put in the hash table it will not be traced. Recall that the traversal algorithm assumes that any object in the hash table has already been traced. Second, when an OOP is computed from the hash value -2, the result is a reference to the object nil. Consequently, all references to clamped objects are converted to nil.

The **writeDict** instance variable plays a role in the clamping of classes. When a class is clamped, its associated write selector is set to **#writeClamped:**. The

writeClamped: method does nothing. This makes it possible to eliminate all the instances of a clamped class without having to explicitly clamp each instance. Consider what happens when some object **obj**, which is an instance of a clamped class, is traced. Assuming **obj** is not in the hash table, the **writeClamped**: method is invoked. **writeClamped**: does not trace **obj** and therefore does not place **obj** into the hash table. When the OOP for **obj** is written, the hash table is again consulted. The partial OOP returned for obj will be **NoRefs** (a class variable with value 0). The OOP computed from the hash value 0 will reference the object **nil**. Consequently, instances of a clamped class are not written to the image and references to them are converted to nil.

A possible source of confusion should be mentioned at this point. The hash word written in each object header is not related to the hash value stored in **hashTable**. The hash word in the object header for a given object is usually constant from one image to the next. In contrast, the hash value in **hashTable** for a given object varies between images. Recall that the hash value is a partial OOP. As such, the hash value depends on the object's location within the image file.

3.2 Sequence of Tracing

Writing a disk image begins when the message

> **writeImage**: roots

is sent to an instance of **SystemTracer**. This method controls the process of writing a disk image by performing the following sequence of tasks:

1. Reserves space in the image file for the image header.
2. Calls **writeSpecial1** to begin tracing the special objects.
3. Calls the method **trace**: to initiate a depth-first search of the object space beginning with each element of **roots** (an array argument). **roots** contains only one element, the **Smalltalk** system dictionary.
4. Calls **writeSpecial2** to complete tracing the special objects.
5. Writes the image header.
6. Closes the image file.

writeSpecial1 performs the following sequence of tasks:

1. Calls **new...write** to write each of the special objects to the image. These calls pass a nil **traceBlock** parameter and an empty **writeBlock** parameter to the **new...write** method. At this point, therefore, objects referenced by the special objects do not get traced. The class OOPs and the non-header data for the special objects do not get written.

2. Initiates a depth-first search (by sending the **trace**: message) beginning with the symbol #**Smalltalk**.

3. Initiates a depth-first search beginning with the symbol #**Processor**.

writeSpecial2 performs the following sequence of tasks:

1. Traces the class and objects referenced by the special objects. This is necessary because tracing was suppressed by **writeSpecial1** (i.e. nil **traceBlock** was passed to **new...write**).

2. Deletes unreferenced symbols from the symbol table. Unreferenced symbols are those that were not reached during traversal of the object space.

3. Traces the symbol table.

4. Writes the class OOP and the non-header data for each special object. This is necessary because writing was suppressed by **writeSpecial1** (i.e. empty **writeBlock** was passed to **new...write**).

5. Writes the OOP for each special object in the known OOP table of the image header.

4 Application Tracer

The application tracer is used to create an image containing only the classes and methods needed to execute some application. To create an image file called **imageName** for some application, evaluate an expression of the form

```
ApplicationTracer writeApplication: aMethodName    "a symbol"
                  in:               aClassName     "a symbol"
                  onFile:           imageName      "a string".
```

Together, the parameters **aMethodName** and **aClassName** identify the initial method. The image written will contain all the classes and methods that might potentially be invoked by the chain of execution started with the initial method.

ApplicationTracer creates a "preserve dictionary" of all the classes and methods needed for a particular application. First, the classes of the special objects are preserved. Recall that special objects are those objects referenced by the Smalltalk interpreter while executing an image. Next, classes that are needed in any image are preserved. These include **Class**, **MetaClass**, **CompiledMethod**, **ByteCodeArray**, **LiteralArray**, **MethodDictionary**, **Process**, and **ProcessorScheduler**. These classes seem to be needed in order for the Smalltalk interpreter to begin running and access the first method to be executed. Each time a class is added to the preserve dictionary all of its superclasses are also added.

Next, the application tracer computes the methods that will potentially be invoked when the initial method is executed (i.e. the method named **aMethodName** in the class named **aClassName**). As discussed in the overview, the methods needed are determined using a graph traversal. Each method visited during the traversal is added to the **preserveDictionary**. I wrote one version of graph traversal for standard Smalltalk and one version for typed Smalltalk.

When the graph traversal completes, **preserveDictionary** contains only the methods and classes needed for the application being traced. The next step is to clamp all of the classes and methods that are not in **preserveDictionary**. Next, global variables that are not needed are clamped. Finally, the **SystemTracer** is invoked to write an image.

4.1 Implementation of Graph Traversal

The untyped and typed versions of the graph traversal algorithm differ in the way they determine the methods needed for an application. In the untyped version, every method that could respond to each message sent must be visited. In the typed version, the number of methods that must be visited is limited by the type of the message receiver. The type identifies a set of classes; only methods in these classes need be visited. In both versions, a message of the form

methodsNeededBy: aSelector **in**: aClass

is sent to an instance of **ApplicationTracer**. This method updates the **preserveDictionary** by adding the classes and methods that are reachable from the initial method. The parameters **aSelector** and **aClass** identify the initial method.

Figure 2 shows the most important methods used in the untyped version of graph traversal. The object **hasBeenReached** is used to store the message selectors that have been added to the preserve dictionary. The object **selectorSet** is initialized with the set of messages sent by the initial method. The recursive method **reachableFrom:with**: determines the methods that can be reached by sending the messages in **selectorSet**. Each time another reachable method is found, **reachableFrom:with**: calls itself with a new **selectorSet**. The pseudocode in figure 3 may be helpful for understanding the **reachableFrom:with**: method.

```
methodsNeededBy: aSelector in: aClass

   | hasBeenReached selectorSet |
   hasBeenReached <- Set new.
   selectorSet <- (aClass compiledMethodAt: aSelector) messages.
   self reachableFrom: selectorSet with: hasBeenReached


reachableFrom: selectorSet with: hasBeenReached

   | messageSet |
   selectorSet do: [:aSelector |
      (hasBeenReached includes: aSelector)
         ifFalse:
            [hasBeenReached add: aSelector.
            (Smalltalk allClassesImplementing: aSelector) do:
               [:aClass |
               self preserveClass: aClass withSelector: aSelector.
               messageSet <-
                  (aClass compiledMethodAt: aSelector) messages.
               self reachableFrom: messageSet with: hasBeenReached]]]
```

Figure 2: Untyped Version of Graph Traversal

```
for each selector s in selectorSet:
   if s has not yet been reached then
      put s into hasBeenReached
      for each class c that implements s:
         preserve c with s
         determine what is reachable from method s in class c
```

Figure 3: Pseudocode for reachableFrom:with: method

```
methodsNeededBy: aSelector in: aClass

   | parseTree aMethod |
   aMethod <- aClass compiledMethodAt: aSelector.
   aMethod class = CompiledMethod
      ifTrue: [self halt: aSelector , ' in ' , aClass name , ' is an
                  untyped method']
      ifFalse:
         [parseTree <- (aClass compiledMethodAt: aSelector) parseTree.
         parseTree postorderDo: [:node |
            self reachableBy: node using: parseTree encoder]]


reachableBy: aParseNode using: anEncoder

   | receiverClasses definingClass key |
   receiverClasses <- aParseNode receiverClasses: anEncoder.
   "receiverClasses is the set of classes to which the message
   represented by aParseNode may be sent."
   receiverClasses do:
      [:class |
      definingClass <-
         class whichClassDefinesSelector: aParseNode selector.
      (self hasBeenReached: aParseNode selector in: definingClass)
         ifFalse:
            [self preserveMethod: aParseNode selector
                  in: definingClass.
            self methodsNeededBy: aParseNode selector
                  in: definingClass]]
```

Figure 4: Typed Version of Graph Traversal

Figure 4 shows definitions used in the typed version of graph traversal. The method **methodsNeededBy:in:** first checks to see if the method identified by **aSelector** in **aClass** is a typed method. If not, an error is generated. It is possible to proceed from this error. The result is that any messages sent by the untyped method will be ignored (i.e. methods invoked by untyped methods will not necessarily be preserved). For typed methods, a postorder traversal of the parse tree is performed. The block passed to **postorderDo:** is evaluated for each node on the parse tree. Consequently, the method **reachableBy:using:** is invoked for each node.

The nodes on the parse tree will be instances of some subclass of Type-**dParseNode**. See [Loy88] for an explanation of parse nodes. Some of these nodes represent message sends and hence invoke methods. The idea is to find these methods that may be invoked and call the **methodsNeededBy:in:** for each one. Nodes that invoke methods will be of class **TypedMessageNode** or class **TypedProcedureNode**. A **TypedMessageNode** or **TypedProcedureNode** responds to the **receiverClasses:** message with the set of classes of the possible message receivers. For each of the possible receiver classes, the method in the class hierarchy that actually implements the message is found. If this method has not previously been reached, it is preserved and a call is made to **methodsNeed-edBy:in:** to determine what additional methods may be invoked by the method just preserved.

When the **receiverClasses:** message is sent to an instance of some subclass of **TypedParseNode** other than **TypedMessageNode** or **TypedProcedureNode**, an empty set is returned. This is accomplished by defining a default implementation (**↑Set new**) for the method **receiverClasses:** in class **TypedParseNode** and overriding this implementation in the classes **TypedMessageNode** and **TypedProcedureNode**. Consequently, for those parse nodes that do not represent message sends, **methodsNeededBy:in:** is not invoked.

4.2 Changes to SystemTracer

I changed **SystemTracer** for the following reasons:

1. To (attempt to) correct (real or imagined) errors in the code.

2. To add debugging or error checking code.

3. To decrease the size of the image created for an application.

4. To add comments.

ApplicationTracer is a subclass of **SystemTracer**. I made changes to **System-Tracer** methods by overriding them in **ApplicationTracer** rather than directly modifying them in **SystemTracer**. Some of the more significant method changes are:

clamp: The **SystemTracer** version did not allow metaclass objects to be clamped; the new version does. See comments in program code.

writeSpecial1 Recall that tracing special objects is a two-step process which begins in **writeSpecial1** and is completed in **writeSpecial2**. I added error checking code to insure that **writeSpecial2** writes the objects in the same positions as **writeSpecial1**. In **writeSpecial1**, a stream instance variable called **specialObjectPositions** is initialized with the file positions where the special objects are written.

writeSpecial2 As additional data is written to the special objects, the file positions are compared with the positions recorded in the instance variable **specialObjectPositions**. If the positions don't match, an error is raised. An image will not work if the special objects have been corrupted. Another change was modifying the code that traced the symbol table. Originally, when a new image was created, all of the symbols were being retained whether needed in the new image or not. I corrected the code so that it would delete unreferenced symbols before writing the symbol table. For this correction, the writing method for the symbol table (**writeIndexable-Pointers:**) also had to be changed.

writeIndexablePointers: I changed this method so that when the symbol table object was traced, it would not trace the symbols. The idea is that a symbol is needed only if it is referenced somewhere besides in the symbol table. Therefore, the symbols needed cannot be identified until the entire application has been traced, at which time all of the symbols that are needed will have been placed in the hash table. Then, as mentioned above, **writeSpecial2** can delete unreferenced symbols (those not in **hashTable**) before writing the symbol table. The code to do this should be understandable if you keep in mind that the symbol table is implemented as an array of arrays.

writeMethodDictionary: I found that a significant amount of space in application images was taken up by nearly empty method dictionaries. I therefore changed this method so that a new method dictionary is created which is just large enough to hold the methods needed for an application. I simplified this method considerably by getting rid of some permutation code which seemed unnecessary. (According to Juanita Ewing at Tektronix, the permutation code is necessary at times).

```
Listing of Methods and Procedures Clamped

Arc -- clamp class

Array -- clamp methods
   (instance)  storeOn: at: printOn: literalType asArray
     hashMappedBy: hash isLiteral at:put:
   (class)

ArrayedCollection -- clamp methods
   (instance)  add: defaultElement storeElementsFrom:to:on:
     size storeOn:
   (class)  with:with:with:with:with: with:with: new with:with:with:
     maxSize with: with:with:with:with: new:withAll:
```

Figure 5: Sample from clamped file

4.3 Finding Out What an Application Image Contains

In addition to the application image file, three other data files can be created during the process of tracing an application. The files are named:

1. <image name>.clamped

2. <image name>.traced

3. <image name>.summary

The clamped file is a listing of clamped classes and methods in alphabetical order by class name. Class methods and instance methods are listed. Figure 5 shows a sample taken from a clamped file.

The traced file contains information printed during the tracing process. Figure 6 shows a sample taken from a traced file. The numbers in square brackets are levels of recursion reached during the tracing of objects. During tracing, the level number is incremented each time **trace:** is invoked and decremented each time an invocation of **trace:** completes. For most objects, only the trace level and class are printed. For instances of class **Symbol**, a pound sign followed by its string representation is printed. For instances of class **String**, the string value is printed in single quotes. For instances of class **CompiledMethod** or **NewCompiledMethod**, the class is printed followed by the class and message selector that identifies the method, if it can be determined.

Trace levels get into the 50's during traversal of even a simple image, so it is very easy to get lost in the traced file. The lines with double stars were an attempt to help. These lines are printed in various methods before and after invoking traces. Inspecting objects in the current image while viewing the file also helps to make the traced file more understandable.

The clamped file and traced file are optional. Before tracing an image, application tracer asks whether or not these files should be produced.

The summary file is always produced when an application is traced. The first part of the file summarizes the objects that were actually traced. Figure 7 shows a sample of this part. Each class that is traced is followed by a pair of numbers. The first is the number of objects of that class that were traced. The second is the total size in bytes taken up by objects of that class. The second part of the summary file lists the classes and methods that were preserved. Figure 8 shows a sample of this part.

5 Image Inspector

Image inspector is used to examine an image that has been saved on disk. The program is invoked by sending a message of the form

ImageInspectorView openOn: 'image file name'.

This message opens a standard system view that displays a list of the special objects. Selecting a special object causes the data for that object to be read from the file and displayed in a text view. In addition, the class object is read and the object that contains the class name is read. The class name of the selected object is displayed along with data from the selected object.

The following is a sample text view for #**doesNotUnderstand**: (one of the special objects):

```
Oop: 8700026C

Count/age/hdr: 7F00
object size (dec.): 30
flags:  RMT=0  CTX=0  type=1
hash word: 54
class: 87000810
   (Symbol)
other bytes: doesNotUnderstand:
(100 111 101 115 78 111 116 85 110 100 101 114 115 116 97 110 100 58 )
```

The object header is displayed in the same format for all objects. The header fields are displayed as hexadecimal values except for object size, which

```
** special object 6 class (SmallInteger class) **
[0]Metaclass
 [1]MethodDictionary
 [1]Array
  [2]'minVal'
  [2]'maxVal'

** special object 6 components **
[0]MethodDictionary
 [1]NewCompiledMethod(SmallInteger test1: )
  [2]LiteralArray
   [3]BytecodeArray
  [2]Set
  [2]Set
  [2]Semaphore
 [1]NewCompiledMethod(SmallInteger test3: )
  [2]LiteralArray
   [3]BytecodeArray
   [3]\#test1:
   [3]\#test2:
  [2]Set
  [2]Set
  [2]Semaphore
[0]Dictionary
 [1]Association
  [2]\#Digitbuffer
  [2]Array
```

Figure 6: Sample from traced file

```
Nbr objects written=1672
File size=50432

Array (472 11736)
Association (139 2780)
BlockContext   (6 984)
BytecodeArray  (16 478)
Character   (256 4096)
CompiledMethod (6 168)
Dictionary  (44 1824)
False (1 12)
Float (34 544)
IdentityDictionary   (4 616)
LinkedList  (9 180)
LiteralArray   (16 536)
Metaclass   (96 4240)
MethodContext  (11 1724)
MethodDictionary  (96 2304)
NewCompiledMethod (10 640)
Process  (3 84)
ProcessorScheduler   (1 20)
Semaphore   (11 264)
Set   (47 3608)
String   (62 1270)
Symbol   (329 6395)
SystemDictionary  (1 4560)
True  (1 12)
UndefinedObject   (1 12)
```

Figure 7: First Part of Summary File

```
Preserved Classes and Methods:

Behavior
BlockZero
   Instance Selectors: value
Boolean
ByteArray
BytecodeArray
Character
False
   Instance Selectors: ifTrue:ifFalse:
Float
IdentityDictionary
Integer
   Instance Selectors: + - <
LinkedList
Magnitude
NewCompiledMethod
Semaphore
SequenceableCollection
Set
SmallInteger
   Instance Selectors: + test1: test3: test2: - <
SystemDictionary
   Instance Selectors: quitPrimitive
True
   Instance Selectors: ifTrue:ifFalse:
Number of methods preserved=13
```

Figure 8: Second Part of Summary File

is displayed as decimal. RMT is the remote object flag; a value of 1 indicates that the object has a remote part. CTX is the context flag; a value of 1 indicates a context object [CW86]. The display format for non-header data depends on the type field. For the example above, type = 1 (byte indexable). The non-header data is therefore displayed as a sequence of bytes. Both the character representation and numeric value is shown for each byte. For type = 2 (16-bit indexable), the data is displayed as a sequence of 16-bit values. For other types, the data is displayed as a sequence of 32-bit values.

The following is a sample text view for the class **String** (one of the special objects). Since type = 0 (non-indexable), the non-header data represents a sequence of instance variables. Notice that the third instance variable is a **SmallInteger** value rather than an OOP. An OOP must be greater than or equal to 80000000 hex.

```
Oop: 8700008C

Count/age/hdr: 7F09
object size (dec.): 48
flags: RMT=0  CTX=0  type=0
hash word: 1C
class: 8700A304
   (String class)
32-bit words (hex):
(87001B34 8700BDB4 1000 8700BDEC 87000000
 80000000 87005104 8700BE1C 87000000)
```

When the text view is active, the middle button menu provides access to objects referenced by the object displayed. Three main options are provided for selecting OOPs. The first option creates a menu of all 32-bit fields in the object displayed. The second option creates a menu showing only the 32-bit fields that represent valid non-nil OOPs. The third menu option allows you to enter or paste in an OOP to be displayed.

When an OOP is selected using any of these three options, a new standard system view is opened to display the selected object. The new window allows the same options to access other objects.

A fourth option is provided for accessing OOPs that are part of dictionary objects. This option provides the capability to select a dictionary key rather than selecting an OOP directly. This option is implemented for objects of class **SystemDictionary** and objects of class **MethodDictionary**. When a key is selected from a **MethodDictionary**, the corresponding value is displayed (which should be an object of class **CompiledMethod** or class **NewCompiledMethod**). When a key is selected from a SystemDictionary, the corresponding association is displayed (which should be an object of class Association).

Using the image inspector to examine an image is somewhat difficult since most of the data in each window comes from a single object. Yet, to understand a single object, one generally needs to look at multiple objects. For example, consider an object that has five 32-bit values in addition to the header data. Suppose you want to to know what the third value represents. It may be a named instance variable. To find its name, you would first display the class object. Then you would have to know that the 5th instance variable of the class object gives an object's instance variables. You would therefore select the 5th OOP and display this object. This object would be an array which contains an object-oriented pointer for each instance variable. If you select the third OOP, you will finally discover the name of the instance variable you were seeking.

The best way to overcome this difficulty is to inspect objects in the current image in parallel with the disk image you are examining. In effect, you use the current image as a "road map" for examining the disk image.

6 Future Work

There are two Smalltalk constructs that my project did not address: blocks and the perform message. A block is an object that contains Smalltalk code. Since they may invoke methods, blocks complicate the process of tracing an application. Consider the following example:

$$\begin{tikzcd}
m1 \arrow[r] \arrow[d] & m2 & \\
G \arrow[r] & B \arrow[r] & m3
\end{tikzcd}$$

In this example, method m1 invokes method m2 and references global object G. Global G references block B which contains code to invoke method m3. If the ApplicationTracer was executed with m1 as the initial method, only methods m1 and m2 would be preserved. Method m3, however, should also be preserved. To correctly handle blocks, the application tracer should trace all objects referenced by a method since these objects may contain blocks.

The perform message complicates tracing because it can be used to invoke any method in a class. Suppose an expression of the following form is evaluated.

```
anObject perform: aSymbol with: anArgument
```

The message sent to **anObject** depends on the value of **aSymbol** which may not be known until run time. It is therefore necessary to preserve every method in **anObject**'s class. Changes are planned for the type system which will allow the number of methods accessible by a perform to be limited.

The images produced from **ApplicationTracer** are larger than they need to be. Several steps could be taken to make images smaller. One would be to eliminate global variables and class variables that are not referenced. This could be done along with determining the methods needed. The **ApplicationTracer methodsNeededBy:in:** method could be changed so that during traversal of the parse tree, the referenced class variables and global variables are identified. Then the clamping process could be modified to clamp class variables and global variables that are not needed.

Another way to decrease the size of the image would be to re-size certain objects when they are traced. As mentioned earlier, this is already being done for method dictionaries. A method dictionary is implemented with two parallel arrays. These arrays are re-sized to be just large enough to hold the methods preserved. The same could be done with the system dictionary. Currently, even though many associations may be removed, the system dictionary size does not change.

7 Conclusion

Developing stand-alone applications using standard Smalltalk is not feasible. Since the language is untyped, there is no way to determine what methods are needed until run-time. Taking the conservative approach of preserving all methods that could possibly be invoked results in an application image which includes most of the Smalltalk environment. Using typed Smalltalk, however, developing stand-alone applications becomes feasible. Type information greatly decreases the number of methods that must be preserved in an application image. The ability to create stand-alone applications is a significant step toward making Smalltalk a more useful language for delivering finished software products.

References

[CW86] Patrick J. Caudill and Allen Wirfs-Brock. A third generation Smalltalk-80 implementation. In *Proceedings of OOPSLA '86, Object-Oriented Programming Systems, Languages and Applications*, pages 119–130, November 1986. printed as SIGPLAN Notices, 21(11).

[JG87] Ralph E. Johnson and Justin O. Graver. *A User's Guide to Typed Smalltalk*. Technical Report, Department of Computer Science, University of Illinois at Urbana-Champaign, 1304 West Springfield, Urbana, Illinois, 1987.

[Loy88] Joseph P. Loyall. *High Level Optimization in a Typed Smalltalk Compiler*. Master's thesis, University of Illinois at Urbana-Champaign, 1988.

[Ung84] David Ungar. Generation scavenging: a non-disruptive high performance storage reclamation algorithm. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 157–167, 1984.